Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 1900 – 1909

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Benchmarking data and compute intensive applications on modern CPU and GPU architectures

Miłosz Ciżnicki[a,*], Michał Kierzynka[a], Piotr Kopta[a], Krzysztof Kurowski[a], Paweł Gepner[b]

[a] *Poznan Supercomputing and Networking Center, Noskowskiego 10 Street, 61-704 Poznan, Poland*
[b] *Intel Corporation, Pipers Way, Swindon Wiltshire SN3 1RJ, United Kingdom*

Abstract

The use of graphics hardware for non-graphics applications has become popular among many scientific programmers and researchers as we have observed a higher rate of theoretical performance increase than the CPUs in recent years. However, performance gains may be easily lost in the context of a specific parallel application due to various both hardware and software factors. Consequently, software benchmarks and performance testing are still the best techniques to compare the efficiency of emerging parallel architectures with the built-in support for parallelism at different levels. Unfortunately, many available benchmarks are either relatively simple application kernels, they have been optimized only for a certain parallel architecture or they do not take advantage of recent capabilities provided by modern hardware and low level APIs. Thus, the main aim of this paper is to present a comprehensive real performance analysis of selected applications following the complex standard for data compression and coding - JPEG 2000. It consists of a chain of data and compute intensive tasks that can be treated as good examples of software benchmarks for modern parallel hardware architectures. In this paper we compare achieved performance results of our standard based benchmarks executed on selected architectures for different data sets to identify possible bottlenecks. We discuss also best practices and advices for parallel software development to help users to evaluate in advance and then select appropriate solutions to accelerate the execution of their applications.

Keywords: benchmarks, GPU, multi-core CPU, JPEG 2000, signal processing

1. Introduction

Hardware vendors always tend to implement their own benchmarks by optimizing algorithms and programming models for next generation processors. It was not and still is not possible to agree on a single benchmark to test all features and capabilities supported by modern processors, especially now with the recent advent of accelerated hardware, in particular Graphics Processing Units (GPUs). Naturally, many efforts have been invested to deal with this problem over the last decades and there are some well-known benchmarking suites available today, e.g. LINPACK [1], LAPACK [2], or NAS Parallel Benchmarks [3]. Unfortunately, those benchmarks have been mostly tailored to measure a system's floating-point computing power using compute intensive procedures rather than data intensive operations.

*Corresponding author
Email address: miloszc@man.poznan.pl (Miłosz Ciżnicki)

doi:10.1016/j.procs.2012.04.208

We believe that data-intensive applications have been increasingly more sophisticated and common operations must be also benchmarked and then considered as an additional evaluation metric.

Our main motivation to develop new benchmarks was a lack of common standard based procedures that have been optimized for emerging hardware architectures. In our opinion, the selected JPEG 2000 standard for data compression and coding consists of various data and compute intensive tasks involving many typical procedures and operations for various scientific applications. Moreover, most of those tasks that are defined by the JPEG 2000 standard have been already optimized by hardware vendors as image and signal processing libraries are core procedures for many third party applications. Due to high porting cost and unavailability of automatic software parallelization tools it is still difficult for many parallel software developers, especially scientific developers, to make a good decision and select an optimal configuration of both hardware and software. We believe that our efforts help some users to understand better often hidden constraints of emerging parallel architectures.

To effectively exploit new capabilities provided by multi-core processors, their modern architectures and low level APIs, software developers must be aware of many factors that will impact the overall performance of their parallel applications. From the hardware perspective, one should take into account not only the clock rate, the number of cores, but also check the memory bandwidth, efficiency of I/O and communication channels, cache topologies, etc. On the other hand, from the software perspective, it has become extremely important to understand data dependencies, data structures and synchronization among multiple parallel tasks for a given problem as all those factors will play a critical role during the execution. In the next sections we will elaborate on all those issues respectively.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 discusses applications used in benchmarks. Section 4 describes various architecture features of two modern parallel compute platforms - CPUs and GPUs. Section 5 includes detailed descriptions of selected benchmarking applications. Section 6 discusses the obtained benchmarking results and provides advices for parallel software development. Section 7 concludes our paper and defines future work.

2. Related works

The progress on efficient parallelization of various algorithms on modern CPUs and GPUs from different fields of science has been published in many our recent papers, e.g. [4, 5, 6, 7, 8]. Furthermore, a number of studies compared the efficiency of GPU and CPU architectures applied by many parallel applications taking into account different measurement criteria. These studies include various computing procedures which provide important foundations for scientific applications. In the context of generic approaches and benchmarks, it is worth mentioning for instance efforts described in [9]. Authors proposed techniques improving GEMM routine in the MAGMA BLAS library for the Fermi GPUs. They compared the improved kernels with MKL, PLASMA and LAPACK on multicore systems. Another good example is FAST [10], which is a kernel that optimizes the number of queries for GPU and multicore CPU architectures. Efforts presented in [11] provided an analysis and efficient implementation of sorting algorithms on CPU and GPU architectures. A comprehensive study presented in [12] revealed new opportunities for the performance improvement of 3D FFT (Fast Fourier transform) library for GPUs. Software based acceleration of DWT (Discrete wavelet transform) on GPUs is concerned in [13, 14, 15, 16], which is useful in the context of image processing. All these example papers show how to port applications onto GPUs by changing algorithms and using new programming models. Some of them include the multicore CPU-optimized versions of the proposed solutions. However, there is still a lack of architectural analysis of CPU and GPU architectures showing various features that have a huge impact on the overall applications performance. Only a few authors addressed this issue, e.g. [17, 18, 19, 20].

The aim of this paper is to present a performance analysis combined with architectural analysis of CPUs and GPUs. Instead of simply showing small or simple application performance tests, we decided to benchmark relatively large both compute and data intensive procedures following the JPEG 2000 standard [21]. The JPEG 2000 standard provides advanced capabilities demanded by more specialized applications, e.g. in the field of medical imaging [22] where lossy compression is not accepted. Additionally, this standard is used in network applications relying on robustness to transmission errors. The JPEG 2000 standard has been successfully used in the context of hyperspectral image compression [23]. However, all the advanced features and high quality compression capabilities yield much higher computational demands.

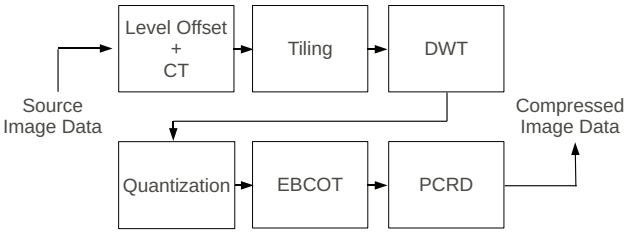


Figure 1: A chain of data and compute intensive tasks in the JPEG 2000 encoding process.

3. Benchmarking applications

The encoding process of JPEG 2000 standard consists of several steps (see Figure 1). At first, the input image components are shifted by level offset to guarantee that all the samples are signed. Then CT (Component Transformation) is applied, which decorrelates components for efficient compression. The result of this stage is a new image in other domain, with the same number of components and samples. Next, as can be observed in the Figure 1, each component is divided into rectangular non-overlapping tiles (tiling process), which are compressed independently, as they are entirely separate images. However, the tiling process is rarely used in practice due to the artifacts produced around the edges of the tiles. Subsequently, DWT (Discrete Wavelet Transform) is used to facilitate the resolution scalability and improve the encoding efficiency by removing the spatial redundancy. The following step is quantization with optional ROI (Region of Interest) coding. The quantized coefficients are grouped into rectangular areas called code-blocks that are encoded independently by the EBCOT (Embedded Block Coding with Optimal Truncation) algorithm [24]. The problem with highly-compressed, entropy coded data is that a few bit errors could completely corrupt the image information. This, in turn, is an issue when JPEG 2000 data is transmitted over a noisy communication channel, so code-blocks typically are of size $32x32$ or $64x64$. Each of these code-blocks is entropy coded separately, which gives a potential for parallelization. The compressed bit-streams of the code-blocks can be divided into a specific number of contiguous segments, or quality layers, by the PCRD-opt (Post-Compression Rate-Distortion Optimization) rate allocation algorithm. The algorithm computes the level of distortion and includes next bytes from code-blocks to the output stream. The main idea is to find, for a given size of target output, the total sum of encoded bytes that minimizes the distortion. In the last part of the encoding chain, the compressed bit-stream is stored in the code-stream.

The Karhunen–Loéve Transform (KLT) is known to be one of the most efficient methods to apply for multi-component image compression and found to improve efficiency in classification and feature extraction process [25]. KLT can be applied as the CT module in the JPEG 2000 compression chain. KLT processing involves calculating covariance matrix of the input signal and then calculating eigenvectors with corresponding eigenvalues.

4. CPU and GPU architectures

Many hardware vendors have developed image processing libraries including JPEG 2000 optimized for their architectures, e.g. Intel IPP [26]. Before we describe application benchmarks and tests to show the overall applications performance let us first describe key architectural features available in recent CPU and GPU.

4.1. Intel CPU

The Intel Xeon X5600 family is the first generation of six core Intel CPUs dedicated to dual socket servers [8]. It is also the first six core processor with integrated memory controller. Intel Xeon X5670 is a 32 nm six cores monolithic die with 12MB of L3 cache, 3 channel integrated memory controller and integrated Quick Path Interconnect interface. Each core in Westmere has a private 32KB first level L1 Instruction and 32kB Data Cache. In addition, the unified 256KB L2 cache is 8 way associative and provides extremely fast access to data and instructions. The L3 Cache is organized as a 16 way set associative, inclusive and shared cache. The L3 cache clock is decoupled from the cores frequency, so theoretically the clock of the cache can be different from the frequency of the cores [27]. Westmere's L3 is implemented as an inclusive cache model. Any cache access misses in the L3 has a guarantee that data is not present it in any of the L2 or L1 caches of the cores. The "core valid" bits mechanism limits unnecessary snoops of the cores during data hit and only checks the identified core where it has a possibility to find a private copy of this

cache line which might be modified in L1 or L2. All these enhancements may have differing performance implications depending on the fields and areas of implementation [28].

The Intel Xeon E5-2660 CPU has improvements on the decoding phase, enhancements to data cache structure and increased instruction level parallelism (ILP). Sandy Bridge single core has 32KB 8-way associative L1 instruction cache and during the decoding phase it is able to decode four instructions per cycle, converting four X86 instructions to micro-ops. To reduce decoding time the new L0 instruction cache for micro-ops has been added to keep 1.5K micro-ops previously decoded. All micro-ops which are already in the L0 cache do not need to be fetched, decoded, and converted to micro-ops again. Consequently, the new L0 cache reduces the power used for these complex tasks by about 80% and improves performance by decreasing decoding latency. A new feature improving the data cache performance is an extra load/store port. By adding the second load/store port Sandy Bridge can handle two loads plus one store per cycle automatically doubling the load bandwidth. One of the methods to achieve the bigger dataflow window is implementing Physical Register File (PRF) instead of centralized Retirement Register File. This method implements a rename register file that is much larger than the logical register file, decreases data duplication and data transfers and finally eliminates movement of data after calculation. This implementation also increases the size of scheduler and number of reorder buffer (ROB) entries by 50% and 3%, respectively and as a result increases the size of the dataflow window globally [29]. All the modifications made to the Sandy Bridge cores are corresponding with changes made in the uncore part of the new CPU. Uncore part integrates last level cache (LLC), the system agent, the memory controller with up to 4 DDR3 memory channels, PCI Express interface generation 3.0, up to 2 QPI links and the DMI. To make this efficient realization Sandy Bridge implements a ring interconnect to connect all these parts and provide faster communication between LLC and system agent area. The cache maintains coherency and ordering for the addresses that are mapped to it. It also maintains “core valid bits”, like the previous generation of Intel Core processors, to reduce unnecessary snoops. Also LLC runs at core frequency and voltage, scaling with the ring and core [30].

4.2. NVIDIA GPU

General-Purpose GPU is a highly parallel, multithreaded, many core processor with very high computational power and memory bandwidth. Fermi is the name for the GF100 architecture that has many expanded capabilities to overcome computational limitations of the previous G80 and GT200 series. The Fermi architecture includes various improvements such as a unified 64-bit memory space, upgraded L1 cache and unified coherent L2 cache. With the unified memory space, pointers can refer to local, shared and global memory locations as well as can be shared among host threads. Each thread in Fermi can push data onto the stack (up to 1KB per thread) and use pop operations to pull data off the stack in a LIFO (Last-In First-Out) fashion. The L1 cache can be configured with 16KB or 48KB of capacity. Additionally, the cached data from global memory can be broadcasted to accelerate irregular memory accesses. There have been made improvements in atomic operations, which are an order of magnitude faster than in the GT200 architecture. The integer operations have been extended to 32bits. In contrast, the GT200 integer ALU (Arithmetic Logic Unit) was limited to 24 bits, as a result mutliple instructions had to be peformed to multiply 32-bit integers. Fermi improves the speed and accurancy of double precision calculations. GF100 provides eight times the peak double precision floating point perfromance over previous architectures.

In our benchmarks we used Tesla S1070 computing system consisting of four GPU processors, it is a variant of the GT200 architecture. For GF100 based GPUs Tesla M2050 and GTX580 cards were chosen. Each Tesla S1070 processor consists of 30 streaming multiprocessors (SMP). An SMP is built from 8 Scalar Processors (SP) cores, two Special Function Units (SFU) and on-chip shared memory with 16KB of capacity. The SP core has the core clock frequency equal to 1.3GHz. One Tesla S1070 processors has 4GB of GDDR3 memory capacity and theoretical bandwidth of 102GB/s. The GF100 based Tesla M2050 card consists of 14 SMPs. Each SMP is built from 32 SPs, four special function units and configurable shared memory with 16KB or 48KB of capacity. Tesla M2050 has 3GB of GDDR5 memory capacity and theoretical bandwidth of 148GB/s. The GTX580 card has similar to Tesla M2050 specification, however it contains 16 SMPs and has less GDDR5 memory (1.5GB) but with higher theoretical bandwidth of 192GB/s.

Table 1: Key hardware architecture features provided by modern CPUs and GPUs.

Name	Cores	Frequency	Mem. Bandwidth	SP Flops	DP Flops	Cache L1/L2/L3	Year
Xeon X5670	6	2.93 GHz	32 GB/s	140.6 GFLOPs	70.3GFLOPs	32 KB/256 KB/12 MB	Q1 2010
Xeon E5 2660	8	2.20 GHz	51 GB/s	281.6 GFLOPs	140.8 GFLOPs	32 KB/256 KB/20 MB	Q1 2012
Tesla S1070 (one GPU)	240	1.3 GHz	102 GB/s	662.1 GFLOPs	77.8 GFLOPs	16 KB/-/-	Q3 2008
Tesla M2050	448	1.15 GHz	148 GB/s	1030.4 GFLOPs	515.2 GFLOPs	16 KB-48 KB/756 KB/-	Q1 2010
GTX 580	512	1.71 GHz	192 GB/s	1581.1 GFLOPs	197.6 GFLOPs	16 KB-48 KB/756 KB/-	Q4 2010

Table 2: JPEG 2000 lossless encoding parameters.

Tiling	Color trans.	Decomp. levels	Wavelet	Codeblock size
no	rev. YUV	4	DWT 5/3	64x64

5. Benchmark details

The following sections describe benchmarking applications that have been selected for real performance testing. The first subsection provides a brief description of our GPU-based implementation and the second one gives an overview of the CPU-based implementations.

5.1. GPU

Our implementation of the JPEG 2000 standard has been developed in the NVIDIA CUDA 4.1 environment. As mentioned in the previous section the JPEG 2000 standard contains several encoding steps, which are performed in a sequential manner. In the first step the level offset is applied on unsigned samples of components. This simple procedure subtracts the same quantity from all the samples and as a result is bound to memory transfer on GPU. In order to obtain high efficiency every thread on GPU is responsible for calculations of several samples. Subsequent encoding step is tiling. However, as image data set easily fits into GPU memory no tiling is used during compression. The next step in the compression process is CT. Although it is an optional step, it may be useful if the image with multiple components is to be compressed. As mentioned, an efficient algorithm for decorrelating hyperspectral data is KLT. The algorithm is based on modified reorthogonalization method of Gram and Schmidt [31]. KLT is applied to the hyperspectral data which include K bands, each of which contains M lines and N samples. At first, a mean vector is calculated: $M_x = [m_1, m_2, \ldots, m_K]^T$, with $m_k = (1/MN)\sum_{i=1}^{M}\sum_{j=1}^{N} x_{i,j}^k$, where $x_{i,j}^k$ is a pixel with spatial coordinates (i, j) in band k. Then, the mean value m_k is subtracted from each band k. Next, the covariance matrix is calculated for each spectral vector: $COV = (1/MN)\sum_{i=1}^{M}\sum_{j=1}^{N} x_{i,j}^{k-1} x_{i,j}^k$. After that, the eigenvalues λ_i and eigenvectors u_i are obtained in the way that satisfies the following formula: $COVu_i = \lambda_i u_i$. As a result, the matrix V is formed with the eigenvectors u_i as columns arranged in descending order of eigenvalue magnitude. Finally, matrix V is used to transform each spectral vector which provides transformed hyperspectral components. All operations are performed on single precision float-point data. In our code CUBLAS (v4.1.28) is used as a parallel implementation of BLAS (Basic Linear Algebra Subprograms) on GPU. After decorrelating image components the next step is DWT in which the tile data is decomposed into horizontal and vertical characteristics. This transform is similar to 1D-DWT in nature, but it is applied in the horizontal (rows) and the vertical (columns) directions which form two-dimensional transform. The component data block is loaded to the shared memory in a way that the processing is done on columns and rows within one kernel invocation including data reordering. Therefore the number of kernel invocations and calls to the global memory is reduced. Once DWT is applied, all the resulting subbands are quantized, which means that the wavelet coefficients are reduced in precision. It involves a few computations only and in consequence each thread is responsible for quantization of 16 samples. After quantization, the integer wavelet coefficients still contain a lot of spatial redundancy. This redundancy is removed by entropy coding (EBCOT) so the data is efficiently compressed into a minimum size bitstream. The process of entropy coding is highly sequential and difficult to parallelize efficiently using many threads. Therefore, each GPU thread does entropy coding on the whole code-block. However, even for small input components there is sufficient amount of work to fill all the multiprocessors with computations. For instance, the input component of size $1920x1080$ with $64x64$ code-blocks size will be spread across $1485/32 = 47$ blocks of threads. The PCRD

algorithm allows to compress image data with a given bitrate. As in the case of entropy coding, each GPU thread calculates distortions within a single code-block, since it requires a small number of computations only. The last step in the compression process is to create and order the packets. This phase consists of creating the progression order and writing the packets to a file. Since this is typically a serial procedure, it is performed on a CPU. More details concerning the implementation can be found in [23] and the source code can be downloaded from [32]. CUJ2K v1.1 [33] is another GPU-based implementation of JPEG 2000 that was considered in our benchmarking experiments.

5.2. CPU

The OpenJPEG 1.5.0 is a library containing a JPEG 2000 codec compliant with Part 1 of the standard. There are also additional modules included in the library, such as compression to the Motion JPEG 2000 format and transmission through the network using JPIP protocol. Kakadu 6.3.1 is a commercial implementation of the JPEG 2000 standard. It allows to compress an image with multiple working CPU threads. Another implementation of the standard is the one included in the Intel IPP (v7.0 build 205.58) library, which also supports multiple execution threads. It should be stressed that none of these CPU-based implementations provides the KLT module for decorrelating components. Therefore, in order to make all the implementations equivalent we have developed KLT on CPU using the MKL library (v10.3), which is a parallel implementation of BLAS. Obviously, to make all the tests fair, this change is explicitly treated in the results.

6. Performance testing and results

6.1. Methodology

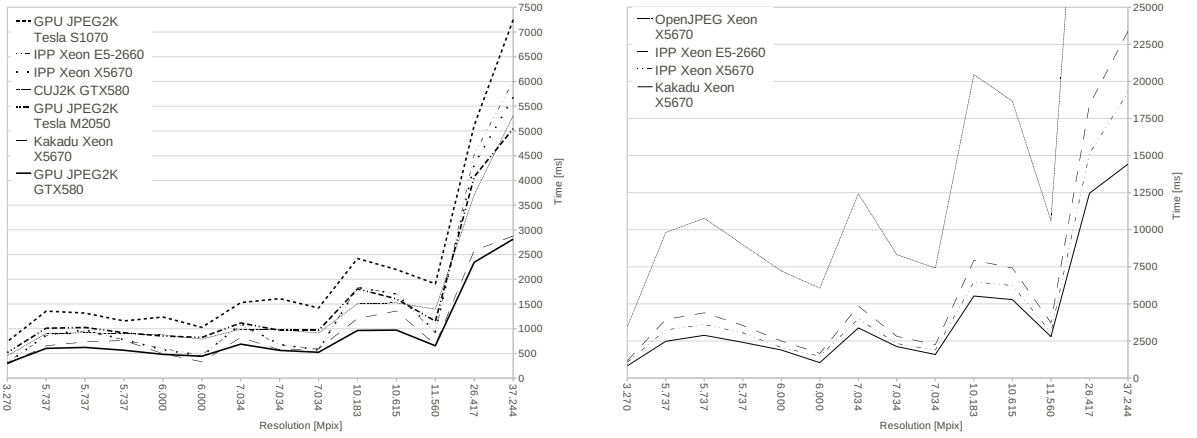
In our study we used dual socket server processors and graphics cards as presented in Table 1. To measure the performance of CPU and GPU processors solely, the data transfer time from RAM to GPU is not included in case of the GPU timings. It is resonable to assume that the image data is available on the device memory as asynchronous transfers during batch processing will hide the data transfer time. However, please note that the data transfer time in the opposite direction (from GPU to RAM) is included in the final results. Also the time needed for reading the input images from the hard disk for both CPU and GPU implementations is omitted. For Kakadu it is worth noting that despite the fact that details regarding instrumentation of the built-in timer functions could not be found, this timer was used for testing purposes. Unfortunately, the parallelization approach in this case is unknown either. The parameters of the JPEG 2000 codec in all cases were set to perform lossless compression, see Table 2. The reason behind this is that in the lossy mode the process of quantization considerably reduces the accuracy of the pixels and thus speedups the encoder. In comparison with the lossless mode, where the quantization is not used, all the information included in the pixels is compressed by the encoder. Therefore, the time needed for lossless compression is the upper bound for its lossy equivalent. The image data set used in the tests was taken from [34]. It contains 8 bits RGB images of various sizes. For the KLT algorithm, the AVAIRIS Cuprite and SubsetWTC scenes were evaluated. These scenes consist of a 350x350-pixels subset with 188 spectral bands (43.92MB) and a 512x640-pixels subset with 224 spectral bands (134.31MB), respectively. Each pixel in the scenes is represented using 16 bits.

6.2. Performance comparison

Figure 2a presents the performance of the JPEG 2000 encoder in case of lossless compression for different processors described in Section 6.1. The result data show that the GPU JPEG2K running on GeForce GTX580 is the fastest implementation. Its average speedup over Kakadu launched on Xeon X5670 machine is 1.1. CUJ2K is about 10% slower than the IPP JEPG2000 implementation, which actually performs similarly on both Xeon 5670 and Xeon E5 2660. It achieves some 70% of the GPU JPEG2K performance. There is one main factor that contributes to the similar efficiency of Xeon 5670 and Xeon E5 2660. The JPEG 2000 part of the IPP library was not optimized for the new features introduced in the Sandy Bridge architecture, especially for the AVX (Advanced Vector Extensions) which is a new extension to the instruction set of processors. In the AVX the length of the SIMD (Single Instruction Multiple Data) registers have been increased to 256 bits, and as a result it can provide up to twice as much performance with respect to 128 bit SIMD extensions. However, the AVX supports currently only floating-point instruction set. Yet, the only part of the algorithm that could potentially benefit here is the DWT in its lossy mode. The support for the integer data will be introduced in the AVX 2 extension in the new Intel Haswell architecture. Nonetheless, the DWT part of

Table 3: Processing times of KLT.

KLT	MKL Xeon E5-2660	MKL Xeon X5670	CUBLAS GTX580
Cuprite	933.64 ms	879.75 ms	798.92 ms
SubsetWTC	3819.40 ms	3183.98 ms	1671.62 ms



(a) (b)

Figure 2: JPEG 2000 lossless compression times (without KLT): a) multi-core CPU and GPU, b) single-core CPU.

JPEG 2000 in IPP is implemented using 128 bit SIMD extensions. It is worth noting that the GPU JPEG2K scales up on the successive GPU architecture very well. On GTX580 it achieves speedups of 2.4 and 1.4 in comparison to Tesla S1070 and Tesla M2050, respectively.

Figure 2b presents the processing times of the JPEG 2000 lossless compression on a single CPU core. The result data show that on Xeon X5670 the Kakadu implementation is on average 1.2 times faster than the IPP implementation. OpenJPEG is about 75% slower than Kakadu. The difference in performance of the IPP JPEG 2000 implementation between Xeon E5 2660 and Xeon X5670 corresponds on average to the clock rates of the processors (1.2X).

Table 3 shows the processing times of the KLT algorithm. For Cuprite hyperspectral dataset the GPU implementation of KLT running on GeForce GTX580 is on average 1.1 times faster comparing to KLT running on Xeon X5670. With larger dataset (SubsetWTC), the difference is even more noticeable in favor to the GPU-based algorithm (1.9X).

6.3. Performance analysis

In this section we analyze the performance results and identify the architectural features that contribute to the performance of each application. The analysis is focused on the IPP implementation and our GPU JPEG2K only, since we could not accurately profile the Kakadu implementation.

6.3.1. Profiling

In the process of profiling the applications it turned out that the most time consuming part of the JPEG 2000 encoder for both IPP and GPU JPEG2K is the EBCOT algorithm, which takes 45% and 85% of the execution time, respectively. It shows that EBCOT is the most demanding and the most difficult part for the parallelization. It is worth noting that in case of GPU JPEG2K the whole encoding process is parallelized on GPU, whereas in IPP only EBCOT is executed in multiple threads. The JPEG 2000 standard defines that during the EBCOT process the image coefficients are grouped into rectangular code-blocks, which can be encoded independently. However, it does not provide enough granularity, especially for GPU, where thread warps should process data in SIMT (Single Instruction Multiple Threads) manner. Generally, the EBCOT code-block coding is composed of bit-plane coder and arithmetic coder. Both of them have inter-coefficients dependencies and a lot of conditional statements, which are unsuitable for the SIMD processing. In our implementation each code-block is processed by one thread. Due to the size of the code-blocks all the data have to be stored in the global memory. Although this approach leads to the performance

improvement, it does not utilize the parallel capabilities of GPU ideally. There are several publications describing methods to speed up the EBCOT compression, e.g. [35], some of them concern the GPU architecture [36, 37] Matela et al. [37] propose a parallel approach to bit plane coding that exploits capabilities of a GPU. As a result, there is some potential for further optimizations in our GPU implementation of EBCOT. Due to the conditional statements EBCOT causes a lot of branch mispredictions in the IPP implementation. This can be partly addressed by using hyperthreading that allows to run two threads on each of the CPU core simultaneously. In this case each thread gets only some part of the resources like cache memory or registers. In our tests on Xeon X5670 and Xeon E5 2660 the hyperthreading was disabled. However, we conducted additional tests on the Core i7 920 processor, where the number of execution threads was doubled with respect to the number of physical cores. As a result, the overall IPP compression time was reduced by about 20%. It can be advantageous to run two threads per a single core in a situation where a large fraction of the time is spent on cache misses and branch mispredictions. However, if any of the core-shared resources is a limitation, then the performance gain is unlikely to occur.

In case of KLT the most time consuming part for both CPU and GPU is the transformation of the spectral vectors using the matrix V, which takes 80% of the time. For Cuprite dataset it needs the same amount of time on CPU as on GPU. However, the computation time increases slower on GPU when the dataset with more bands is processed. Likewise, the calculation of covariance matrix scales better on a GPU when larger hyperspectral data set is used. It shows that the GPU performance is dependent on the scale of the problem. Thus, in order to efficiently exploit the massive parallelism of GPUs and effectively use the hardware capabilities, the problem itself needs to scale up properly, such that thousands of threads are defined and used in computations.

6.3.2. Cache and bandwidth

In GPU JPEG2K code-block data used in the EBCOT processing are stored in the global memory. The Fermi architecture allows to configure the local memory, so it can be partitioned to favor either the shared memory or L1 cache. As the code-block data do not fit in the shared memory anyway, we could partition local memory to favor the L1 cache. However, this implies that the cache line is 128-byte long. In order to enable 32-byte memory access one needs to disable the L1 cache completely. EBCOT accesses data in striped manner with a nominal height of four coefficients. The method of accessing the memory in such a way does not allow to efficiently use 128-byte cache line and hence most of the global memory bandwidth is wasted. As a result, disabling the L1 cache provides the performance gain of about 5%.

In the case of CPU implementation the cache can alleviate the pressure of limited external memory bandwidth, since the code-block data entirely fit in it. Xeon E5-2660 provides three levels of caches, each of which with increasing size: 32KB (L1), 256KB (L2) and 20MB (L3). Both L1 and L2 caches are provided per core, but L3 is shared by all the cores. At the same time, GeForce GTX 580 has 16KB/48KB of L1 cache per a single multiprocessor and 756KB of L2 cache shared by all multiprocessors. The L1 cache in GPU is designed only for the spatial but not temporal use, as is the case with CPU. Thus on CPU the EBCOT is compute bound and the performance scales up with increasing computing speed.

6.3.3. Synchronization

CPU provides a memory consistency model together with a cache coherency protocol. Due to the fact that cache coherence is not available on GPU, we have to assure that computation is divided into independent working sets, otherwise synchronization barriers need to be used. In our first implementation of DWT on GPU the processing was done on columns and rows in separate kernel invocations. As a result, there was a global synchronization barrier between these kernel invocations, which ensured that data from all levels of memory hierarchy was saved to the global memory. However, a more efficient approach was to divide computations into similar-sized independent patches, even though additional margin data was required here.

6.3.4. Computations

The second most time consuming part of the JPEG 2000 standard both in IPP and GPU JPEG2K is Tier-2, where the resulting data are ordered and packed into codestream. In a single core implementations, such as OpenJPEG, the most computational intensive part beside EBCOT is DWT. However, DWT can be easily parallelized by using vector extensions, since the data can be processed simultaneously. As described in previous section, DWT in IPP is parallelized using 128 bit SIMD extensions. As a result, in IPP the DWT algorithm takes about 8% of the whole

processing time, whereas in GPU JPEG2K it takes only about 2%. This is close to the 6X memory bandwidth ratio of GeForce GTX580 to Xeon E5 2660.

6.3.5. Floating-point precision

The iterative GS (Gram and Schmidt) algorithm estimates the eigenvalues for a given precision ε. The number of iterations that the algorithm performs depends on expected maximum error ε requirement. During the calculation of the eigenvalues the number of iterations on GPU is almost equal to CPU. What is more, one iteration takes on average 4 times more computation time on GPU, as in each iteration a relatively small dataset is utilized. In order to compare the floating-point precision between matrix V transformation on CPU and GPU, an MSE (Mean Square Error) was calculated and was equal to 0.0005. As differences between CPU and GPU calculations, such as the number of concurrent threads participating in blocking of floating-point matrix, can affect the final result [38], the obtained value seems to be reasonable.

6.3.6. Development effort

Since high efficiency does not come for free, algorithms running on a GPU must be parallelized as well as balanced properly. Development of a GPU-based algorithm may cost some extra time and effort to understand and use the appropriate programming model, a model that may not be consistent with the simple idea of scalar processor with automatic cache protocol. However, the analysis and programming techniques used to develop GPU algorithms can be helpful while writing multicore applications as well. Nevertheless, the architecture specific software optimization is essential to fully utilize available resources of both CPU and GPU.

7. Conclusions

In this paper, we presented a set of new standard based data and compute intensive benchmarks on CPUs and GPUs. We showed that currently a lot of development efforts must be assumed for a specific parallel software optimization on both CPU and GPU architectures. However, we have identified key capabilities that should be used to improve significantly the application performance on modern multi-core CPUs, especially multithreading, cache blocking and use of vector extensions. On the other hand, parallel applications can benefit a lot using GPUs by using local memory and registers, minimizing the number of divergent warps and trying to overlap memory transactions with computations. Our analysis of the optimized benchmarks on recent CPU and GPU platforms using complex procedures defined by the JPEG 2000 standard confirmed that even for one application some internal parts fit much better to a GPU architecture whereas other parts can be executed more efficiently on multi-core CPUs. Therefore, in our opinion both CPUs and GPUs should be considered for many scientific applications as complementary hybrid solutions. It is clear for us that it is worth to invest many efforts in the preparatory phase before starting implement or port any parallel software on emerging processors. Due to high software porting costs and unavailability of automatic software parallelization tools decisions must be made carefully based on various performance evaluation criteria. Some of them together with basic guidelines have been presented in this paper to help readers better understand the complexity of the performance analysis problem.

8. Acknowledgments

This work was supplied in part by Polish Ministry of Science and Higher Education Grant COST 805 Open Network for High-Performance Computing on Complex Environments. The authors gratefully acknowledge the help and support of Jamie Wilcox from Intel EMEA Technical Marketing HPC Lab.

References

[1] J. Dongarra, P. Luszczek, A. Petitet, The LINPACK Benchmark: past, present and future, Concurrency and Computation: Practice and Experience 15 (9) (2003) 803–820.

[2] E. Anderson, Z. Bai, C. Bischof, LAPACK Users' guide, Vol. 9, Society for Industrial Mathematics, 1999.

[3] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, et al., The NAS parallel benchmarks summary and preliminary results, in: Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on, IEEE, 1991, pp. 158–165.

[4] M. Blazewicz, S. Brandt, M. Kierzynka, K. Kurowski, B. Ludwiczak, J. Tao, J. Weglarz, CaKernel - a parallel application programming framework for heterogenous computing architectures, Scientific Programming 19 (4) (2011) 185–197.
[5] M. Ciznicki, M. Kierzynka, K. Kurowski, B. Ludwiczak, K. Napierala, J. Palczynski, Efficient isosurface extraction using marching tetrahedra and histogram pyramids on multiple GPUs, LNCS, To appear.
[6] J. Blazewicz, W. Frohmberg, M. Kierzynka, E. Pesch, P. Wojciechowski, Protein alignment algorithms with an efficient backtracking routine on multiple GPUs, BMC Bioinformatics 12:181 (181).
[7] J. Blazewicz, W. Frohmberg, M. Kierzynka, P. Wojciechowski, G-MSA – GPU-based, fast and accurate algorithm for multiple sequence alignment, Journal of Parallel and Distributed Computing, To appear.
[8] P. Kopta, M. Kulczewski, K. Kurowski, T. Piontek, P. Gepner, M. Puchalski, J. Komasa, Parallel application benchmarks and performance evaluation of the Intel Xeon 7500 family processors, Procedia Computer Science 4 (2011) 372–381.
[9] R. Nath, S. Tomov, J. Dongarra, An improved MAGMA GEMM for Fermi GPUs, Innovative Computing Laboratory, University of Tennessee, Tech. Rep.
[10] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, P. Dubey, FAST: fast architecture sensitive tree search on modern CPUs and GPUs, in: Proceedings of the 2010 international conference on Management of data, ACM, 2010, pp. 339–350.
[11] N. Satish, C. Kim, J. Chhugani, A. Nguyen, V. Lee, D. Kim, P. Dubey, Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort, in: Proceedings of the 2010 international conference on Management of data, ACM, 2010, pp. 351–362.
[12] A. Nukada, S. Matsuoka, Auto-tuning 3-D FFT library for CUDA GPUs, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, 2009, p. 30.
[13] J. Matela, GPU-based DWT acceleration for JPEG2000, in: Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, 2009, pp. 136–143.
[14] W. van der Laan, A. Jalba, J. Roerdink, Accelerating wavelet lifting on graphics hardware using CUDA, IEEE Trans. Parallel Distrib. Syst 22 (1) (2011) 132–146.
[15] D. Bader, V. Agarwal, S. Kang, Computing discrete transforms on the Cell Broadband Engine, Parallel Computing 35 (3) (2009) 119–137.
[16] J. Franco, G. Bernabé, J. Fernández, M. Acacio, A parallel implementation of the 2D wavelet transform using CUDA, in: Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on, IEEE, 2009, pp. 111–118.
[17] M. Anderson, B. Catanzaro, J. Chong, E. Gonina, K. Keutzer, C. Lai, M. Murphy, D. Sheffield, B. Su, N. Sundaram, Considerations when evaluating microprocessor platforms, in: Proceedings of the 3rd USENIX conference on Hot topic in parallelism, USENIX Association, 2011, pp. 1–1.
[18] V. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, et al., Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU, in: ACM SIGARCH Computer Architecture News, Vol. 38, ACM, 2010, pp. 451–460.
[19] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, A. Shringarpure, On the limits of GPU acceleration, in: Proceedings of the 2nd USENIX conference on Hot topics in parallelism, USENIX Association, 2010, pp. 13–13.
[20] J. Jenkins, I. Arkatkar, J. Owens, A. Choudhary, N. Samatova, Lessons learned from exploring the backtracking paradigm on the GPU, Euro-Par 2011 Parallel Processing (2011) 425–437.
[21] Information technology - JPEG 2000 image coding system: Core coding system (2004).
[22] D. Foes, E. Mukab, R. Sloneb, B. Erickson, M. Flynr, D. Clunie, K. Lloyd Hildebrand, S. Younga, JPEG 2000 compression of medical imagery, in: Proc. of SPIE, Vol. 3980, 2002.
[23] M. Ciznicki, K. Kurowski, A. Plaza, GPU implementation of JPEG2000 for hyperspectral image compression, in: Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, Vol. 8183, 2011, p. 12.
[24] D. Taubman, High performance scalable image compression with EBCOT, Image Processing, IEEE transactions on 9 (7) (2000) 1158–1170.
[25] B. Penna, T. Tillo, E. Magli, G. Olmo, Transform coding techniques for lossy hyperspectral data compression, Geoscience and Remote Sensing, IEEE Transactions on 45 (5) (2007) 1408–1421.
[26] S. Taylor, Optimizing Applications for Multi-core Processors: Using the Intel Integrated Performance Primitives, Intel Press, 2007.
[27] K. Barker, K. Davis, A. Hoisie, D. Kerbyson, M. Lang, S. Pakin, J. Sancho, A performance evaluation of the Nehalem quad-core processor for scientific computing, Parallel Processing Letters 18 (4).
[28] P. Gepner, M. Kowalik, D. Fraser, K. Wackowski, Early performance evaluation of new Six-Core Intel® Xeon® 5600 family processors for HPC, in: Parallel and Distributed Computing (ISPDC), 2010 Ninth International Symposium on, IEEE, 2010, pp. 117–124.
[29] Intel Advanced Vector Extensions Programming Reference (July 2009).
[30] P. Gepner, V. Gamayunov, D. L. Fraser, Evaluation of Executing DGEMM Algorithms on modern Multicore CPU, in: In Proceedings of The Parallel and Distributed Computing and Systems 2011 Conference, 2011.
[31] M. Andrecut, Parallel GPU implementation of iterative pca algorithms, Journal of Computational Biology 16 (11) (2009) 1593–1599.
[32] GPU JPEG2K, https://apps.man.poznan.pl/trac/jpeg2k/.
[33] A. Weiß, M. Heide, S. Papandreou, N. Fürst, CUJ2K.
[34] Test images, http://www.imagecompression.info/test_images/.
[35] J. Chiang, C. Chang, C. Hsieh, C. Hsia, High efficiency EBCOT with parallel coding architecture for JPEG2000, EURASIP journal on applied signal processing 2006 (2006) 17–17.
[36] S. Datla, N. Gidijala, Parallelizing motion JPEG 2000 with CUDA, in: Computer and Electrical Engineering, 2009. ICCEE'09. Second International Conference on, Vol. 1, IEEE, 2009, pp. 630–634.
[37] J. Matela, V. Rusnak, P. Holub, GPU-Based Sample-Parallel Context Modeling for EBCOT in JPEG2000, in: Sixth Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'10)–Selected Papers, Vol. 16, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 77–84.
[38] N. Whitehead, A. Fit-Florea, Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs, rn (A+ B) 21 (2011) 1–1874919424.